无锁的概念

      在谈论无锁概念时，总会关联起乐观派与悲观派，对于乐观派而言，他们认为事情总会往好的方向发展，总是认为坏的情况发生的概率特别小，可以无所顾忌地做事，但对于悲观派而已，他们总会认为发展事态如果不及时控制，以后就无法挽回了，即使无法挽回的局面几乎不可能发生。这两种派系映射到并发编程中就如同加锁与无锁的策略，即加锁是一种悲观策略，无锁是一种乐观策略，因为对于加锁的并发程序来说，它们总是认为每次访问共享资源时总会发生冲突，因此必须对每一次数据操作实施加锁策略。而无锁则总是假设对共享资源的访问没有冲突，线程可以不停执行，无需加锁，无需等待，一旦发现冲突，无锁策略则采用一种称为CAS的技术来保证线程执行的安全性，这项CAS技术就是无锁策略实现的关键，下面我们进一步了解CAS技术的奇妙之处。

无锁的执行者-CAS

CAS

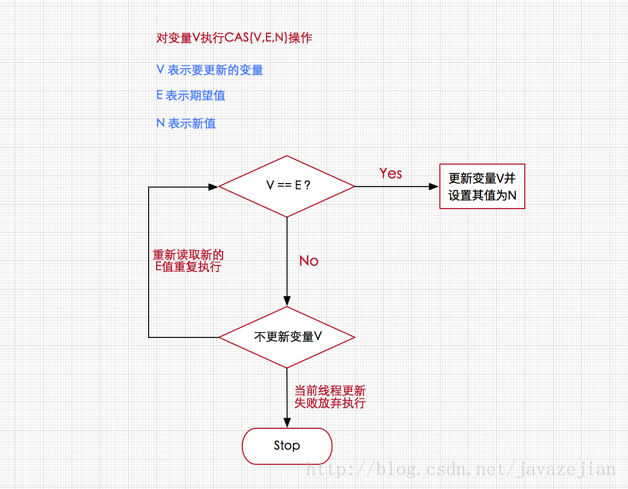
CAS的全称是Compare And Swap 即比较交换，其算法核心思想如下

执行函数：CAS(V,E,N)

其包含3个参数

V表示要更新的变量 E表示预期值 N表示新值

如果V值等于E值，则将V的值设为N。若V值和E值不同，则说明已经有其他线程做了更新，则当前线程什么都不做。通俗的理解就是CAS操作需要我们提供一个期望值，当期望值与当前线程的变量值相同时，说明还没线程修改该值，当前线程可以进行修改，也就是执行CAS操作，但如果期望值与当前线程不符，则说明该值已被其他线程修改，此时不执行更新操作，但可以选择重新读取该变量再尝试再次修改该变量，也可以放弃操作，原理图如下



由于CAS操作属于乐观派，它总认为自己可以成功完成操作，当多个线程同时使用CAS操作一个变量时，只有一个会胜出，并成功更新，其余均会失败，但失败的线程并不会被挂起，仅是被告知失败，并且允许再次尝试，当然也允许失败的线程放弃操作，这点从图中也可以看出来。基于这样的原理，CAS操作即使没有锁，同样知道其他线程对共享资源操作影响，并执行相应的处理措施。同时从这点也可以看出，由于无锁操作中没有锁的存在，因此不可能出现死锁的情况，也就是说无锁操作天生免疫死锁。

CAS是cpu的原子指令是连续的。

**Unsafe类**

Unsafe类存在于sun.misc包中，其内部方法操作可以像C的指针一样直接操作内存，单从名称看来就可以知道该类是非安全的，毕竟Unsafe拥有着类似于C的指针操作，因此总是不应该首先使用Unsafe类，Java官方也不建议直接使用的Unsafe类，但我们还是很有必要了解该类，因为Java中CAS操作的执行依赖于Unsafe类的方法，注意Unsafe类中的所有方法都是native修饰的，也就是说Unsafe类中的方法都直接调用操作系统底层资源执行相应任务，关于Unsafe类的主要功能点如下：

**内存管理，Unsafe类中存在直接操作内存的方法；**

|  |
| --- |
| //第一个参数o为给定对象，offset为对象内存的偏移量，通过这个偏移量迅速定位字段并设置或获取该字段的值，  //expected表示期望值，x表示要设置的值，下面3个方法都通过CAS原子指令执行操作。  public final native boolean compareAndSwapObject(Object o, long offset,Object expected, Object x);    public final native boolean compareAndSwapInt(Object o, long offset,int expected,int x);    public final native boolean compareAndSwapLong(Object o, long offset,long expected,long x); |

|  |
| --- |
| 将一个线程进行挂起是通过park方法实现的，调用 park后，线程将一直阻塞直到超时或者中断等条件出现。unpark可以终止一个挂起的线程，使其恢复正常。Java对线程的挂起操作被封装在 LockSupport类中，LockSupport类中有各种版本pack方法，其底层实现最终还是使用Unsafe.park()方法和Unsafe.unpark()方法  //线程调用该方法，线程将一直阻塞直到超时，或者是中断条件出现。  public native void park(boolean isAbsolute, long time);    //终止挂起的线程，恢复正常.java.util.concurrent包中挂起操作都是在LockSupport类实现的，其底层正是使用这两个方法，  public native void unpark(Object thread); |

Unsafe是单例的

# 并发包中的原子操作类(Atomic系列)

自旋转

自旋锁是一种假设在不久将来，当前的线程可以获得锁，因此虚拟机会让当前想要获取锁的线程做几个空循环(这也是称为自旋的原因)，在经过若干次循环后，如果得到锁，

就顺利进入临界区。如果还不能获得锁，那就会将线程在操作系统层面挂起，这种方式确实也是可以提升效率的。但问题是当线程越来越多竞争很激烈时，

占用CPU的时间变长会导致性能急剧下降，因此Java虚拟机内部一般对于自旋锁有一定的次数限制，可能是50或者100次循环后就放弃，直接挂起线程，让出CPU资源。

|  |
| --- |
| public class SpinLock {  private AtomicReference<Thread> sign =new AtomicReference<>();    public void lock(){  Thread current = Thread.currentThread();  while(!sign .compareAndSet(null, current)){  }  }    public void unlock (){  Thread current = Thread.currentThread();  sign .compareAndSet(current, null);  }  } |